Where We Are

Source code

if (b == 0) a = b;

Low-level IR code

Optimizations

Optimized Low-level IR code

Assembly code

cmp $0,%rcx
cmovz %rax,%rdx
Low IR to Assembly Translation

• Low IR code (TAC):
  • Variables (and temporaries)
  • No run-time stack
  • No calling sequences
  • Some abstract set of instructions

• Translation
  • Calling sequences:
    • Translate function calls and returns
    • Manage run-time stack
  • Variables:
    • globals, locals, arguments, etc. assigned memory location

• Instruction selection:
  • map sets of low level IR instructions to instructions in the target machine

t3 = this.x
t3 = t2 * t3
t0 = t1 + t2
r = t0
t4 = w + 1
k = t4
x86-64 crash course

• a.k.a. CS 240 review, upgrade to 64 bits
• Focus on specific recurring details we need to get right.

• Calling Conventions

• Memory addressing
  • Field access
  • Array indexing

• Wacky instructions
  • Division
  • Store absolute address
  • setCC and movzbq
### x86 IA-32: registers

#### General Purpose Registers
- `%eax` (equivalent to `%ax`, `%ah`, `%al`)
- `%ecx` (equivalent to `%cx`, `%ch`, `%cl`)
- `%edx` (equivalent to `%dx`, `%dh`, `%dl`)
- `%ebx` (equivalent to `%bx`, `%bh`, `%bl`)
- `%esi` (equivalent to `%si`)
- `%edi` (equivalent to `%di`)
- `%esp` (equivalent to `%sp`)
- `%ebp` (equivalent to `%bp`)

#### 16-bit Virtual Registers
(Backwards compatible)

- `%ah` (high byte of `%al`)
- `%ch` (high byte of `%cl`)
- `%dh` (high byte of `%dl`)
- `%bh` (high byte of `%bl`)

#### High/Low Bytes of Old 16-bit Registers
- `accumulate`
- `counter`
- `data`
- `base`
- `source index`
- `destination index`
- `stack pointer`
- `base pointer`
x86-64: more registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

64-bits wide

Only %rsp is special-purpose.
Most 2-operand instructions

\texttt{movq Source, Dest:}

• Get argument(s) from \textit{Source} (and \textit{Dest} if, \textit{e.g.}, arithmetic)

• Store result in \textit{Dest}.

• Operand Types:
  
  • \textbf{Immediate}: Literal integer data, starts with $\$
    • Examples: $\$0x400$ or $\$-533$ or $\$\text{foo}$
  
  • \textbf{Register}: One of 16 integer registers
    • Examples: \%\text{rax} or \%\text{rsi}
  
  • \textbf{Memory}: 8 consecutive bytes in memory, at address held by register
    • Simplest example: (\%\text{rax})
    • Various other “address modes”
Memory Addressing Modes

• General Form: $D(Rb,Ri,S) \quad \text{Mem}[\text{Reg}[Rb] + S*\text{Reg}[Ri] + D]$
  - $D$: Displacement (offset): literal value represented in 1, 2, 4, or 8 bytes
  - $Rb$: Base register: Any register
  - $Ri$: Index register: Any register except $\%\text{rsp}$
  - $S$: Scale: literal 1, 2, 4, or 8

• Special Cases: use any combination of $D$, $Rb$, $Ri$ and $S$
  - $(Rb)$ $\quad \text{Mem}[\text{Reg}[Rb]] \quad (Ri=0,S=1,D=0)$
  - $D(Rb)$ $\quad \text{Mem}[\text{Reg}[Rb] + D] \quad (Ri=0,S=1)$
  - $(Rb,Ri,S)$ $\quad \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]] \quad (D=0)$
  - $D(,Ri,S)$ $\quad \text{Mem}[S*\text{Reg}[Ri]+D] \quad (Rb=0)$

...
Big Picture: Memory Layout

Stack variables

- Previous fp
- Local 0
- ... Local n
- Return address
- Param 0
- ... Param n

Heap variables

Global variables

- Global 0
- ... Global n
(A) x86 IA-32/Linux Stack Frames

- **Return Address**
- **Saved Registers**
- **Local Variables**
- **Callee Argument 0**
- **Callee Argument n**

**Stack Registers**
- **Base/Frame pointer** %ebp
- **Stack pointer** %esp

**Caller's base pointer**

**Caller Frame**

**Callee Frame**

**Stack Top**

**High addresses**

**Low addresses**
(A) x86 IA-32/Linux Stack Frames

- Base/Frame pointer \%ebp
- Stack pointer \%esp

- Stack Registers
- Arguments for next call
- Saved Registers + Local Variables
- Return Address
- Callee Argument 0
- Callee Argument n
- Stack Top

- High addresses
- Caller Frame
- Callee Frame
- Low addresses
(B) x86-64 with old-style Stack Frames

\[
x = 16 + n \times 8 \quad x(\%rbp)
\]

- \(24(\%rbp)\)
- \(16(\%rbp)\)
- \(8(\%rbp)\)
- \(0(\%rbp)\)
- \(-8(\%rbp)\)
- \(-16(\%rbp)\)

Stack Registers
- Base/Frame pointer \(\%rbp\)
- Stack pointer \(\%rsp\)

Caller's base pointer

Saved Registers + Local Variables

Callee Argument \(n\)

Callee Argument 0

Return Address

Arguments for next call

Stack Top

High addresses

Callee Frame

Low addresses

Caller Frame
x86-64/Linux ABI
No base pointer
1st 6 args in registers
Stack access relative to %rsp
Compiler knows frame size

(C) x86-64 with new-style Stack Frames

128-byte red zone
safe between calls

Stack pointer %rsp

Compiler knows frame size
(C) Typical x86-64 **new-style Stack**

**x86-64/Linux ABI**
- No base pointer
- 1<sup>st</sup> 6 args in registers
- Stack access relative to `%rsp`
- Compiler knows frame size
(D) x86-64 with **mixed-style Stack**

No base pointer

**All args on stack**

Stack access relative to %rsp

Compiler knows frame size

---

Stack pointer %rsp

---

High addresses

Low addresses

Callee Frame

Caller Frame

Stack Top
Saving Registers During Function Calls

• **Problem:** execution of callee may overwrite necessary values in registers

• **Possibilities:**
  • Callee saves and restores registers
  • Caller saves and restores registers
  • ... or both
### x86-64/Linux ABI: register conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rax</code></td>
<td>Return value</td>
</tr>
<tr>
<td><code>%rbx</code></td>
<td>Callee saved</td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td>Argument #4</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>Argument #3</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>Argument #2</td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td>Argument #1</td>
</tr>
<tr>
<td><code>%rsp</code></td>
<td>Stack pointer</td>
</tr>
<tr>
<td><code>%rbp</code></td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%r8</code></td>
<td>Argument #5</td>
</tr>
<tr>
<td><code>%r9</code></td>
<td>Argument #6</td>
</tr>
<tr>
<td><code>%r10</code></td>
<td>Caller saved</td>
</tr>
<tr>
<td><code>%r11</code></td>
<td>Caller saved</td>
</tr>
<tr>
<td><code>%r12</code></td>
<td>Callee saved</td>
</tr>
<tr>
<td><code>%r13</code></td>
<td>Callee saved</td>
</tr>
<tr>
<td><code>%r14</code></td>
<td>Callee saved</td>
</tr>
<tr>
<td><code>%r15</code></td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

Only `%rsp` is special-purpose.
ICC Calling Convention

- Always follow **x86-64/Linux register save convention**.

- To interface with **external code** (LIB), use:
  - **(C)** x86-64/Linux calling convention.

- To interface with other **ICC-generated code**, use one of:
  - **(B)** use frame pointer and stack pointer, all args on stack
    - Easiest, more work to convert if you convert to (C) later.
  - **(D)** use only stack pointer, all args on stack
    - Moderately easy, easier to convert to (C) later.
  - **(C)** x86-64/Linux calling convention
    - Harder, requires more register allocation work, more efficient, only use this later if you have time.
Example (B)

• Consider call $\text{foo}(3, 5)$:
  • %rcx caller-saved
  • %rbx callee-saved
  • result passed back in %rax

• Code before call instruction:
  
  push %rcx  # push caller saved registers
  push $5   # push second parameter
  push $3   # push first parameter
  call _foo # push return address & jump to callee

• Prologue at start of function:
  
  push %rbp  # push old fp
  mov %rsp, %rbp # compute new fp
  sub $24, %rsp # push 3 integer local variables
  push %rbx  # push callee saved registers
Example (B)

• Epilogue and end of function:

  ```
  pop %rbx             # restore callee-saved registers
  mov %rbp,%rsp       # pop callee frame, including locals
  pop %rbp            # restore old fp
  ret                 # pop return address and jump
  ```

• Code after call instruction:

  ```
  add $16,%rsp        # pop parameters
  pop %rcx            # restore caller-saved registers
  # %rax contains return result
  ```

You are not likely to need to save/restore registers with the most basic code generation techniques.
Simple Code Generation ($D$)

• Three-address code makes it easy to generate assembly
  
  e.g. \( a = p+q \)  
  
  \[
  \begin{align*}
  \text{movq } & 16(\%rsp), \%rax \\
  \text{addq } & 8(\%rsp), \%rax \\
  \text{movq } & \%rax, 24(\%rsp)
  \end{align*}
  \]

• Need to consider many language constructs:
  • Operations: arithmetic, logic, comparisons
  • Accesses to local variables, global variables
  • Array accesses, field accesses
  • Control flow: conditional and unconditional jumps
  • Method calls, dynamic dispatch
  • Dynamic allocation (new)
  • Run-time checks
Division

movq ..., %rcx # divisor, any reg. but %rax,%rdx
movq ..., %rax # dividend
cqto # sign-extend %rax into %rdx:%rax
idivq %rcx # divide %rdx:%rax by %rcx
    # quotient in %rax
    # remainder in %rdx
String Literals, using calling convention (D)

.rodata

...  
.align 8  
.quad 13
strlit3:
    .ascii "Hello, World!"
...
.text
...

# t4 = "Hello, World!"
# Works on both LLVM/Mac OS X and GCC/Linux:
leaq strlit3(%rip), %rax  # GCC only: movq $strlit3, %rax
movq %rax, 8(%rsp)
# Library.println(t4);
movq 8(%rsp), %rax
movq %rax, -8(%rsp)
subq 8, %rsp
callq __LIB_println

Method vectors/vtables and vtable pointer initialization will be similar.
cmpq and testq

cmpq %rcx,%rax computes %rax - %rcx,
sets CF, OF SF, ZF, discards result

testq %rax,%rcx computes %rax & %rcx,
sets SF, ZF, discards result

Flags/condition codes:
  CF: carry flag, 1 iff carry out
  OF: overflow flag, 1 iff signed overflow
  SF: sign flag, 1 iff result's MSB=1
  ZF: zero flag, 1 iff result=0

Common pattern to test for 0 or <0: testq %rax, %rax
### jmp and jCC

#### Always jump
- `jmp` 1 Unconditional
- `je, jz` ZF Equal / Zero
- `jne, jnz` ~ZF Not Equal / Not Zero

#### Jump iff condition
- `jg` ~ (SF ^ OF) & ~ZF Greater (Signed)
- `jge` ~ (SF ^ OF) Greater or Equal (Signed)
- `jl` (SF ^ OF) Less (Signed)
- `jle` (SF ^ OF) | ZF Less or Equal (Signed)
- `js` SF Negative
- `jns` ~SF Nonnegative
- `ja` ~CF & ~ZF Above (unsigned)
- `jb` CF Below (unsigned)

<table>
<thead>
<tr>
<th>jCC</th>
<th>Condition</th>
<th>Jump iff ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je, jz</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne, jnz</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>jg</td>
<td>~ (SF ^ OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~ (SF ^ OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF ^ OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF ^ OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>ja</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
**setCC** and **movzbxq**

```assembly
# t7 = t4 <= t9
movq 72(%rsp), %rdx      # %rdx = t9
cmpq 32(%rsp), %rdx      # set flags: t9 - t4
setle %al                 # set byte to 0x00 or 0x01
                            # based on condition le: <=
                            # as in %rdx <= %rcx
movzbxq %al, %rax         # move, zero-extend byte to quad
                            # (Extend to 64 bits.)
movq %rax, 56(%rsp)        # t7 = result
```

Set has all the same flavors as conditional jump.
Accessing Heap Data

- Heap data allocated with new (Java) or malloc (C/C++)
  - Allocation function returns address of allocated heap data
  - Access heap data through that reference

- Array accesses in Java
  - access `a[i]` requires:
    - computing address of element: `a + i * size`
    - accessing memory at that address
  - Indexed memory accesses do it all
  - Example: assume size of array elements is 8 bytes, and local variables `a`, `i` (offsets –8, -16)

```plaintext
a[i] = 1

mov -8(%rbp), %rbx  (load a)
mov -16(%rbp), %rcx  (load i)
mov $1, (%rbx,%rcx,8) (store into the heap)
```
Run-time Checks

• **Run-time checks:**
  • Check if array/object references are non-null
  • Check if array index is within bounds

• **Example: array bounds checks:**
  • if v holds the address of an array, insert array bounds checking code for v before each load (...=v[i]) or store (v[i] = ...)
  • Array length is stored just before array elements:

```
cmp $0, -24(%rbp)                (compare i to 0)
jl ArrayBoundsError             (test lower bound)
mov -16(%rbp), %rcx             (load v into %ecx)
mov -8(%rcx), %rcx              (load array length into %ecx)
cmp -24(%rbp), %rcx             (compare i to array length)
jle ArrayBoundsError            (test upper bound)
...
```
Object Layout

- **Object consists of:**
  - Methods
  - Fields

- **Layout:**
  - Pointer to VT, which contains pointers to methods
  - Fields.

```
stack) layout (static data) (code)
  p  vptr  |  getx  |  getx
       x    |     gety
       y    |  code
```

```vptr
x
y```
Field Offsets

- Offsets of fields from beginning of object known statically, same for all subclasses

```java
class Shape {
    Point LL /* 8 */ , UR; /* 16 */
    void setCorner(Point p);
}

class ColoredRect extends Shape {
    Color c; /* 24 */
    void setColor(Color c);
}
```
Field Alignment

• In many processors, a 32-bit load must be to an address divisible by 4, address of 64-bit load must be divisible by 8
• x86: unaligned access typically permitted, but slower
• Fields should be aligned

```
struct {
    int x;
    char c;
    int y;
    char d;
    int z; double e;
}
```
VTable Lookup

C <: B <: A

A     f
|     |
B     f,g,h
|     |
C     f,g,h,e

class A {
    void f() {...} 0
}
class B extends A {
    void f() {...} 0
    void g() {...} 1
    void h() {...} 2
}
class C extends B {
    void e() {...} 3
}
VTable Layouts

- Index of f is the same in any object of type T <: A

- To execute a method m:
  - Lookup entry m in vector
  - Execute code pointed to by entry value
Code Generation: Virtual Tables

• Statically allocate one vtable per class

.data
ListVT: .quad _List_first
       .quad _List_rest
       .quad _List_length
Method Arguments

• Receiver object is (implicit) argument to method

```java
class A {
    int f(int x, int y) {
        ... }
}
```

compile as

```java
int f(A this, int x, int y) {
    ... }
```
Code Generation: Method Calls

• Pre-function-call code:
  • Save registers
  • Push parameters
  • call function by its label

• Pre-method call:
  • Save registers
  • Push parameters
  • Push receiver object reference
  • Lookup method in vtable
Example

```c
o.foo(2, 3);
```

```assembly
push $3
push $2
push %rax
mov (%rax), %rbx
call *8(%rbx)
add $24, %rsp
```

The compiler knows the offset of `foo` in the table.
Interfaces, Abstract Classes

• Interfaces
  • no implementation
  • no dispatch vector info
  • (slow lookup a la SmallTalk)

• Abstract classes are halfway:
  • define some methods
  • leave others unimplemented
  • no objects (instances) of abstract class
  • Can construct vtable- just leave abstract entries "blank"
Code Sharing

- Don’t actually have to copy code...
Code Generation: Library Calls

• Pass params in registers
  • %rdi for first param
  • %rsi for second param

• Return result is in %rax

• Warning: library functions may modify caller save registers

```assembly
movq $100, %rdi
call __LIB_printi
...

movq $20, %rdi
call __LIB_random
movq %rax, -32(%rbp)
```
Code Generation: Allocation

- Heap allocation: `o = new C()`
  - Allocate heap space for object
  - Store pointer to vtable into newly allocated memory

```
movq $32, %rdi  # 3 fields + vptr
call __LIB_allocObject
leaq _C_VT(%rip), %rdi
movq %rdi, (%rax)
```